Outline

- the Collection interface
- parametrized classes and interfaces
- the Collection hierarchy
- Sets and Maps
- the AbstractCollection hierarchy

Collection Interface

- **a collection holds values of a different type E**
- **the number of values can change over time**
  - unless the collection is unmodifiable
- **collections typically have at least two constructors**
  - one with no arguments
  - one with a collection as an argument, makes a copy of the collection
  - these constructors are not defined by the interface
    - since interfaces don’t list constructors!
  - all collections in the Java standard library include these two constructors

Collection Methods

- **methods:**
  - `size`
  - search: `contains`, `containsAll` (why not containsAny?)
  - remove is also a search operation: `remove`, `removeAll`, `removeIf`, `retainAll`
  - `add`, `addAll`
  - `toArray`, `iterator`, `spliterator`, `stream`, `parallelStream`
- **Collection does not specify an ordering of elements**
  - so a set class or set interface can implement Collection
  - e.g. the interface `java.util.Set<E>`
- **in comparison, a List imposes an ordering (0, 1, 2, 3, …) on its elements**

Type Parameters

- **classes and interfaces can be parametrized over types**
- **we have seen this for the specific Collections ArrayList and LinkedList**
- **in this case, the type parameter:**
  - outside the class, must be provided whenever an object of that class is created

```
LinkedList<String> strings = new LinkedList<>();
ArrayList<String> string2 = new ArrayList<String>();
List<Integer> ints = new ArrayList<>();
```

  - inside the class or interface, stands for the actual type that the caller will use

```
public class LinkedList<E> {
  LinkedNode<E> head;
  ...
    E item;
```

- **inside the class or interface, the type parameter represents any object type, so all the Object methods are available, including `toString` and `equals`**

Type Parameters in Practice

- **type parameters are conventionally written with a single upper-case letter:**
  - E for element or T for type
  - this makes it easy to tell where code is using a type parameter rather than an actual type
  - the compiler does not enforce this convention
- **classes and interfaces can be parametrized over multiple types:**
  - public interface keyValueStore<K, V> { ...
- **the Java type system is not strong enough to let us safely create an array of a parametrized type:**

```
@SuppressWarnings("unchecked")
public ArrayList() {
    data = (E []) new Object [16];
}
```

Collection Hierarchy

- **List and Set are sub-interfaces of Collection**
  - Lists are ordered
  - Sets are unordered and cannot have duplicates
- **Many classes implement collection, including ArrayList, Vector, LinkedList, Stack, and at least 5 Set classes**
- **Because there are many possible constraints on adding values, add returns false if the element cannot be added**
  - specifically if adding to a set that already has the value
  - if refusing to add for any other reason, add throws an exception
    - e.g. if adding null to a collection that will not store null values

Sets

- **a set is a collection of elements where:**
  - each element is unique
    - `.equals` is used to determine element uniqueness
    - the value of set elements should never be modified, otherwise the uniqueness property may be violated
  - elements are unordered
    - this is different from lists and from all your homeworks
    - iterators are free to return elements in any order

Set Interfaces

- **a Set is a collection of elements where:**
  - each element is unique
  - elements are unordered
- **a SortedSet keeps elements in order as defined by a comparator**
- **a NavigableSet is a SortedSet that allows searching for the next element before or after a given value**
  - the given value need not be in the set
  - for example, navigableSet.lower("hello world") gives the nearest string preceding "hello world" even if "hello world" is not in the set
  - descendingIterator() gives an iterator that returns the elements in reverse order

Map Interface

a map is similar to an array or array list, but instead of using an integer index, values are indexed by arbitrary objects of type K:

```
interface Map<K, V> {
  V get(Object key); // throws ClassCastException if key is not of type K
  V put(K key, V value);
  V remove(Object key);
  V replace(K key, V value);
  Set<K> keySet();   // a Set represents unique values, and keys are unique
  Collection<V> values();
}
```

- Map does not inherit from Collection and is parametrized on two types
- keys must be unique, objects need not be
- it is often convenient to use strings as keys

Abstract Collection Classes

- **to create a Collection:**
  - extend `AbstractCollection` and implement `add`, `size`, and `iterator`
  - or extend `AbstractList` and implement `add`, `get`, `remove`, `set`, and `size`
  - or extend `AbstractSequentialList`, and implement `listIterator` and `size`
- **these abstract classes let you create a collection with a minimum of work**
  - this strategy is not allowed for your homeworks!
  - in-class exercise: why not?

`AbstractCollection`

- **to create a Collection, extend `AbstractCollection` and implement (override) `add`, `size`, and `iterator`**
  - no need to override the `add` method if the class is unmodifiable
- **since `Collection` has no `get` method, these three methods are sufficient for implementing a reasonable collection**
- **individual implementations may override additional methods of `Collection`**

AbstractList

- **extends `AbstractCollection`**
- **to create a `Collection`, extend `AbstractList`, and implement `add`, `get`, `remove`, `set`, and `size`**
- **this abstract list is generally intended for collections where elements are stored in a "random access" data structure such as an array**
  - `get`, `set` are typically O(1)
  - `add`, `remove` may be O(1) at the end and O(n) elsewhere
  - `set` is only needed for modifiable lists, `add` and `remove` are only needed for variable-sized lists
    - so you can have an unmodifiable, fixed-sized list by providing only `get` and `size`!!!
  - the `AbstractList` class uses `get` and `size` to implements the iterators
- **elements are stored at a specific index**

`AbstractSequentialList`

- **extends `AbstractList`**
- **to create a `Collection`, extend `AbstractSequentialList`, and implement `listIterator` and `size`**
- **this class is designed for sequential-access data structures such as doubly-linked lists**
- **`listIterator` is powerful enough to add and remove and find elements**
  - but the `get` operation then takes time O(n)
- **implementing the `listIterator` might be as hard as implementing the list methods directly!**
  - but if you already have a `listIterator`, the other methods are free